Workgroup: Network Working Group
Internet-Draft: draft-taylor-uuid-ncname-01
Updates: RFC4122 (if approved)
Published: 15 January 2021
Intended Status: Informational
Expires: 19 July 2021
Authors: D. Taylor
Independent

Compact UUIDs for Constrained Grammars

Abstract

The Universally Unique Identifier is a suitable standard for, as the name suggests, uniquely identifying entities in a symbol space large enough that the identifiers do not collide. Many formal grammars, however, are too restrictive to permit the use of UUIDs in their canonical representation (described in RFC 4122 and elsewhere), despite it being useful to do so. This document specifies an alternative compact representation for UUIDs that preserves some properties of the canonical form, with three encoding varietals, to fit these more restrictive contexts.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The formal grammar production "one or more letters or underscores followed by zero or more letters, digits, or underscores" (denoted by the regular expression /^[A-Za-z_][0-9A-Za-z_]*$/) is ubiquitous in computing. It is often used for identifiers, and for good reasons. We may encounter some variations on this theme, like admitting hyphens, dots, or Unicode alphanumerics. Some systems may impose additional constraints, like case-sensitivity (or the lack of it), explicit upper- or lower-case letters, or limits on identifier length.

UUIDs are standardized 128-bit identifiers with many useful properties, and there are many places where it would make sense to use them, but their canonical representation, either with or without the URN prefix (see RFC 4122 [RFC4122]) does not conform to the constraint described above:

*UUIDs contain hyphens (and colons in the case of URNs),

*UUIDs potentially start with a digit,

*UUIDs are potentially too long for the slot.

This leads to developers creating incompatible, ad-hoc solutions. The goal of this specification is to address an ostensible need for a UUID representation that is fewer characters in length than the canonical form, and that always starts with a letter.

This document specifies a strategy for a compact representation of UUIDs, with three encoding variants, as well as the related transformations to and from the familiar UUID format. The proposed name for the general strategy is *UUID-NCName*, after the NCName production [XML-NAMES], which is pervasive in XML and RDF applications. The encodings are thus styled as *UUID-NCName-32*, *UUID-NCName-58*, and *UUID-NCName-64*, referring to the base of their respective encodings. Each encoding presents tradeoffs in alphabet, symbol length, and case sensitivity.

1.1. Requirements

The aim of this specification is to eliminate work on the part of developers who find themselves in the position of needing to squeeze UUIDs into the aforementioned grammars, by defining alternative representations that are:

*Significantly shorter lexically than the canonical UUID representation (even after removing the hyphens),

*Guaranteed to begin with with a letter (/^[A-Za-z]/),

*Deployable (through different encodings) in case-sensitive and case-insensitive contexts,

*Devoid of non-payload characters (i.e., every character in the representation is part of the UUID; except for any padding to a prescribed length; see Section 3),

*Fully isomorphic to the canonical UUID representation (i.e., accommodates all possible future UUID versions and variants that [RFC4122] does),

*Amenable to detection and identification by heuristic (Section 4.1) (in a manner analogous to the canonical UUID representation).

1.2. Motivation & Applications

The purpose of an identifier in general is to pick out some information resource or other, such that it can be referred to, ideally unambiguously. The purpose of a large, generated identifier like the UUID, is to satisfy the uniqueness criterion while also specifying a datatype and normal form for said identifiers, and ultimately alleviate the need to sit down and think these identifiers up. Why one would want to go inserting UUIDs in places they wouldn't otherwise fit, is so these UUIDs can be cross-referenced in some other database where they *do* fit. Consider:

* A programming environment that separates the task of writing logic from naming things, stores identifiers internally as UUID-NCName-32 prior to transforming them on display or export, thus preserving the correctness of the syntax.
* A component content management system that uses UUIDs to identify elementary content components, uses the UUID-NCName-64 (or UUID-NCName-58, but in this case Base64 works too and is one byte shorter) representations of the same UUIDs as fragment identifiers for when those components are transcluded.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Strategy

Not all 128 bits of a UUID are data; rather, several bits are masked. The top four bits of the third segment, known as time_hi_and_version, specify the UUID's version, which is fixed. Up to three high bits in the following segment, called clock_seq_hi_and_reserved, specify the variant: how the UUID - if applicable - is meant to be read. We remove these masked quartets (we round up to four bits for the variant) and use them as "bookends" for the rest of the identifier, mapping them to the first sixteen symbols of the Base32 table [RFC4648], which are all letters. These "bookend" characters provide an analogous hint to a developer of the nature of the UUID, just as one can by looking at the third and fourth segments of a canonical hexadecimal UUID representation.

The remaining 120 bits, which we bit-shift to close the gaps of the two masked quartets we removed, now divide evenly by both 5 and 6, the number of bits per character in Base32 and Base64, respectively. Base58 [Base58] encoding cannot map to an even number of bits, but we don't have the same concerns with regard to padding as we do with Base32 and Base64. Indeed with Base58 we have a different padding issue: some inputs yield shorter outputs than others, so we pad the Base58 representation with underscore characters (_, a character *not* in the Base58 alphabet) to get a consistent length. The details are laid out in the encoding algorithm (Section 5.1) below.

The transformation takes a UUID such as 068d0f22-7ce5-4fe2-9f81-3a09af4ed880, and returns the results:

* ea2gq6it44x7c7aj2bgxu5weaj for Base32,

* EBdYYqP7vH96E8SLjJaTH_J for Base58, and

* EBo0PInzl_i-BOgmvTtiAJ for Base64.

These symbols will always start and end with case-insensitive letters (/^[A-Za-z]/), and the entire Base32 symbol is case-insensitive.

4. Syntax

Here is the ABNF grammar for the productions uuid-ncname-32, uuid-ncname-58, and uuid-ncname-64:

```
uuid-ncname-32 = bookend 24base32 bookend

uuid-ncname-58 = bookend base58 bookend

uuid-ncname-64 = bookend 20base64url bookend

bookend        = %x41-50 / %x61-70 ; [A-Pa-p]

base32         = %x32-37 / %x41-5a / %x61-7a ; [2-7A-Za-z]

b58char        = %x31-39 / %x41-48 / %x4a-4e / %x50-5a /
                 %x61-6c / %x6d-7a ; [1-9A-HJ-NP-Za-km-z]

base58         = 15b58char 6"_" / 16b58char 5"_" /
                 17b58char 4"_" / 18b58char 3"_" /
                 19b58char 2"_" / 20b58char "_" / 21b58char
                 ; (symbol sequence plus appropriate padding)

base64url      = %x2d / %x30-39 / %x41-5a / %x5f / %x61-7a
                 ; [-0-9A-Z_a-z]
```

"Bookends" are 4-bit sequences (nybbles, quartets, etc.) which we map directly onto the Base32 table from [RFC4648]. Indeed the this portion of the Base64 table is identical, though we say Base32 to underscore the fact that bookend characters are case-insensitive. Certain environments encode meaning into the case of the first character of a symbol, so it is important that its literal representation be flexible. There is likewise little value in arbitrarily constraining the last character. Nevertheless, UUID-NCName-32 symbols **SHOULD** be generated entirely lower-case, while

UUID-NCName-58 and UUID-NCName-64 symbols **SHOULD** be generated with the bookend characters in upper-case.

4.1. Detection Heuristic

All encodings of UUID-NCName are a fixed length:

- UUID-NCName-32 is always 26 bytes.
- UUID-NCName-58 is always 23 bytes.
- UUID-NCName-64 is always 22 bytes.

All encodings likewise use the same "bookend" mechanism which always correspond to the first 16 symbols of Base32 (A to P, with the side effect that they are effectively case-insensitive). The first and last character in all three representations will therefore always be the same, modulo case, for a given UUID. Furthermore, since these "bookend" characters represent the version and variant bits, they will correspond to predictable values. Version 4 (random) UUIDs, for instance, will always begin with E, and any UUID with its variant bits set as defined in RFC 4122 [RFC4122] will always terminate (again, modulo case) with I, J, K, or L.

Given these facts, any UUID-NCName representation **MAY** be captured (and its "bookends" separated) using the following regular expression:

```
/\b([A-Pa-p]) # zero-width boundary and version bookend

([2-7A-Za-z]{24}|[-0-9A-Z_a-z]{20}| # base32 and 64

  (?:[1-9A-HJ-NP-Za-km-z]{15}_{6}|[1-9A-HJ-NP-Za-km-z]{16}_{5}|
     [1-9A-HJ-NP-Za-km-z]{17}_{4}|[1-9A-HJ-NP-Za-km-z]{18}___|
     [1-9A-HJ-NP-Za-km-z]{19}__|[1-9A-HJ-NP-Za-km-z]{20}_|
     [1-9A-HJ-NP-Za-km-z]{21})) # base58 with underscore pad

([A-Pa-p])\b/x # variant bookend and zero-width boundary
```

The scrupulous may also wish to examine the bookend characters, for which the first should only correspond to the numbers 1 through 5 (plus zero for the nil UUID) for UUID versions known at the time of this writing, and the other should have the same bits set as expected in Section 4.1.1 of RFC 4122 [RFC4122]. Note however that there is room in the spec for another ten UUID versions (up to a hypothetical version 15), and another variant bit that is currently unused.

This detection method is considered a heuristic because it is possible to identify false-positive matches in random strings of text, just as it would be with a canonical UUID representation. It is assumed that there would be sufficient enough context to positively identify these alternative UUID representations in the wild.

4.2. Equivalency

Two UUID-NCName symbols are necessarily identical if they convert to the same (canonical) UUID. Two UUID-NCName-32 symbols are identical if their string values match when normalized to all upper- or lower-case letters. Two UUID-NCName-58 or UUID-NCName-64 symbols are identical if their string values match when the "bookend" characters are normalized to either upper- or lower-case.

5. Algorithms

These are candidate algorithms for encoding and decoding the symbols, transforming them to and from the canonical UUID representation. Equivalent algorithms no doubt exist, but these are the ones used in the reference implementations (Appendix B).

5.1. Encoding Algorithm

First we apply the shifting algorithm:

1. Convert the UUID to a binary string bin.

2. Convert bin to an array of four 32-bit unsigned network-endian integers ints.

3. Extract version as (ints[1] & 0x0000f000) >> 12.

4. Extract variant as (ints[2] & 0xf0000000) >> 24.

5. Assign ints[1] = (ints[1] & 0xffff0000) | ((ints[1] & 0x00000fff) << 4) | ((ints[2] & 0x0fffffff) >> 24).

6. Assign ints[2] = (ints[2] & 0x00ffffff) << 8 | (ints[3] >> 24).

7. Assign ints[3] = (ints[3] << 8) | variant.

8. Convert ints back into a binary string and return it along with the version.

Then apply one of the formatting algorithms; here is Base32:

1. Take the binary string bin and shift the last octet to the right by one bit.

2. Encode bin with the Base32 algorithm to get the string b32.

3. Truncate b32 to 25 characters, removing any padding.

4. Convert version to its value in the Base32 table.

5. Return version concatenated to b32, optionally in either upper or lower case.

And Base58:

1. Remove the last octet from the binary string bin, convert it to an integer and assign it to variant.

2. Shift variant to the right by 4 bits, and convert it to its value in the Base32 table.

3. Encode the remaining bin with the Base58 algorithm to get the string b58.

4. If b58 is less than 21 characters long, append underscores (_) until it is.

5. Convert version to its value in the Base32 table.

6. Return the concatenation of version, b58, and variant.

And finally, Base64:

1. Take the binary string bin and shift the last octet to the right by two bits.

2. Encode bin with the base64url algorithm to get the string b64.

3. Truncate b64 to 21 characters, removing any padding.

4. Convert version to its value in the Base32 table.

5. return version concatenated to b64.

5.2. Decoding Algorithm

1. First use the detection heuristic (Section 4.1) to determine whether the symbol ncname is Base32, Base58, or Base64.

2. Remove the first character of the symbol ncname and convert it into an integer according to the Base32 spec; call that integer version.

3. If ncname is Base58:

    a. Remove the last character and decode it to an integer according to the Base32 spec; call that integer variant.

    b. Shift variant four bits to the left.

    c. Remove all trailing underscores from the remainder of ncname.

    d. Decode the remainder of ncname with the Base58 algorithm as bin.

    e. Append the octet corresponding to the value of variant to bin.

4. Otherwise:

    a. If ncname is Base64, and the last character is lowercase, set it to uppercase.

    b. Append padding if necessary to satisfy the decoder, A====== for Base32 and A== for Base64.

    c. Decode the remainder of ncname by either the base32 or base64url decoding algorithm into binary string bin.

    d. If ncname is Base32, shift the last octet of bin one bit to the left; if Base64 shift it two bits.

Now we apply the shifting algorithm in reverse:

1. Ensure version &= 0xf so it is in the range of 0-15.

2. Convert the binary string bin into an array of four 32-bit unsigned network-endian integers ints.

3. Assign variant = (ints[3] & 0xf0) << 24.

4. Shift and assign ints[3] >>= 8.

5. Union and assign ints[3] |= ((ints[2] & 0xff) << 24).

6. Shift and assign ints[2] >>= 8.

7. Union and assign ints[2] |= ((ints[1] & 0xf) << 24) | variant.

8. Assign ints[1] = (ints[1] & 0xffff0000) | (version << 12) | ((ints[1] >> 4) & 0xfff).

9. Convert ints back into the new binary string bin.

10. Format bin as a canonical UUID.

6. IANA Considerations

There are no discernible IANA considerations associated with this specification.

7. Security Considerations

As UUID-NCName symbols are isomorphic to their canonical UUID representations, the security considerations for these symbols also the same as [RFC4122], though we repeat here the admonition not to assume that UUIDs are hard to guess.

8. Normative References

[Base58] Nakamoto, S. and M. Sporny, "The Base58 Encoding Scheme", 31 October 2020, <https://tools.ietf.org/html/draft-msporny-base58-02>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

9. Informative References

[XML-NAMES] Bray, T., Hollander, D., Layman, A., Tobin, R., and H S. Thompson, "Namespaces in XML 1.0 (Third Edition)", 8 December 2009, <https://www.w3.org/TR/2009/REC-xml-names-20091208/>.

Appendix A. Samples

Version	Canonical UUID Representation
0, Nil	00000000-0000-0000-0000-000000000000

Version	Canonical UUID Representation
1, Timestamp	ca6be4c8-cbaf-11ea-b2ab-00045a86c8a1
2, DCE "Security"	000003e8-cbb9-21ea-b201-00045a86c8a1
3, MD5	3d813cbb-47fb-32ba-91df-831e1593ac29
4, Random	01867b2c-a0dd-459c-98d7-89e545538d6c
5, SHA-1	21f7f8de-8051-5b89-8680-0195ef798b6a

Table 1: Samples of canonical UUID representations

Version	Base32	Base64
0, Nil	aaaaaaaaaaaaaaaaaaaaaaaaaa	AAAAAAAAAAAAAAAAAAAAAA
1, Timestamp	bzjv6jsglv4pkfkyaarninsfbl	BymvkyMuvHqKrAARahsihL
2, DCE "Security"	caaaah2glxepkeaiaarninsfbl	CAAAD6Mu5HqIBAARahsihL
3, MD5	dhwatzo2h7mv2dx4ddykzhlbjj	DPYE8u0f7K6Hfgx4Vk6wpJ
4, Random	eagdhwlfa3vm4rv4j4vcvhdlmj	EAYZ7LKDdWcjXieVFU41sJ
5, SHA-1	feh37rxuakg4jnaabsxxxtc3ki	FIff43oBRuJaAAZXveYtqI

Table 2: Samples of UUID-NCName-32 and UUID-NCName-64 representations

Version	Base58
0, Nil	A111111111111111______A
1, Timestamp	B6fTkmTD22KpWbDq1LuiszL
2, DCE "Security"	C11KtP6Y9P3rRkvh2N1e__L
3, MD5	D2ioV6oTr9yq6dMojd469nJ
4, Random	E3UZ99RxxUJC1v4dWsYtb_J
5, SHA-1	Fx7wEJfz9eb1TYzsrT7Zs_I

Table 3: Samples of UUID-NCName-58 representations

Appendix B. Implementations

As of this writing, there are two implementations of UUID-NCName:

*Perl, https://metacpan.org/pod/Data::UUID::NCName

*Ruby, https://rubygems.org/gems/uuid-ncname

Author's Address

Dorian Taylor
Independent

Email: ietf@doriantaylor.com
URI: https://doriantaylor.com/